Real-time voxel rendering algorithm based on Screen Space Billboard Voxel Buffer with Sparse Lookup Textures

Szymon Jabłoński
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
s.jablonski@ii.pw.edu.pl

Tomasz Martyn
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
martyn@ii.pw.edu.pl

ABSTRACT

In this paper, we present a novel approach to efficient real-time rendering of numerous high-resolution voxelized objects. We present a voxel rendering algorithm based on triangle rasterization pipeline with screen space rendering computational complexity. In order to limit the number of vertex shader invocations, voxel filtering algorithm with fixed size voxel data buffer was developed. Voxelized objects are represented by sparse voxel octree (SVO) structure. Using sparse texture available in modern graphics APIs, we create a 3D lookup table for voxel ids. Voxel filtering algorithm is based on 3D sparse texture ray marching approach. Screen Space Billboard Voxel Buffer is filled by voxels from visible voxels point cloud. Thanks to using 3D sparse textures, we are able to store high-resolution objects in VRAM memory. Moreover, sparse texture mipmaps can be used to control object level of detail (LOD). The geometry of a voxelized object is represented by a collection of points extracted from object SVO. Each point is defined by position, normal vector and texture coordinates. We also show how to take advantage of programmable geometry shaders in order to store voxel objects with extremely low memory requirements and to perform real-time visualization. Moreover, geometry shaders are used to generate billboard quads from the point cloud and to perform fast face culling. As a result, we obtained comparable or even better performance results in comparison to SVO ray tracing approach. The number of rendered voxels is limited to defined Screen Space Billboard Voxel Buffer resolution. Last but not least, thanks to graphics card adapter support, developed algorithm can be easily integrated with any graphics engine using triangle rasterization pipeline.

Keywords

Computer graphics, voxel rendering, sparse voxel octree, sparse texture, point cloud, geometry shader, billboarding

1 INTRODUCTION

Voxel representations and rendering algorithms are one of the most extensively studied subjects in the field of computer graphics. For many years, voxels have been used in the visualization and analysis of medical and scientific data such as MRI scans [Potts04]. Nowadays, voxels representations are widely used in many fields of computer science, engineering and computer graphics, with applications ranging from fluid simulation to digital sculpting tools. However, because of high memory consumption and rendering complexity, their usage was limited to non-real-time graphics engines.

Thanks to increased computation power of today's GPUs and newly developed techniques, it seems that voxel-based representations are ready for real-time applications. Cyril Crassin was able to perform visualization of global illumination based on sparse voxel octree (SVO) and voxel cone tracing [Crassin11]. There are also a few promising implementations of efficient ray tracing of SVO [Laine10] and even object animation and deformation in real-time [Bau11, Wil13].

Computing performance is one of the most important measurements of real-time computer graphics algorithms. SVO ray tracing implementations showed that in this case ray tracing approach is much faster than triangle rasterization. Ray tracing rendering is scalable with screen resolution with the fixed cost of rendering, independent of the virtual scene complexity. Unfortunately, ray tracing does not have direct support from graphic accelerators and popular graphics APIs.

In the case of triangle rasterization pipeline, it is easy to overfill vertex shader invocations by redundant geometry data. We developed an algorithm which solves

that problem for voxel visualization with triangle rasterization pipeline. By selecting render candidates and filling a fixed size buffer prior to rendering process, we are able to limit voxel shader invocations to a defined maximum number. Thanks to that we achieved comparable or even better rendering performance results in comparison with SVO ray tracing approach. Moreover, we gain the support of modern hardware graphics APIs and rendering algorithms. Last but not least, the developed algorithm can be easily integrated with any popular game engines and used in video games.

2 RELATED WORK

There is a wide selection of literature on visualizing voxel objects. Over the years, many methods of direct and indirect voxel rendering have been developed. However, only a few of them are actually using a polygonal representation of the voxel structure in rendering process. We will focus on papers that are most directly related to our work.

One of the oldest and most citied method is Marching Cubes, presented by Lorensen and Cline in 1987 [Lorensen87]. The idea is to extract a polygonal mesh of an isosurface from a 3D discrete scalar field. Marching Cubes implementations are mainly used in the field of medical visualizations and special effects with what is usually called metasurfaces. There are also a few improvements of the base algorithm, like dual contouring [Ju02]. However, the marching cubes approach does not generate satisfying results in visualization of voxelized 3D objects with a lot of textures, like normal maps, ambient occlusion maps etc.

Splatting is one of the most studied methods of direct volume rendering. It was originally introduced by Westover [Westover89] The basic algorithm projects each voxel to the screen and integrate it into an accumulating render target. Using a painter's algorithm, it solved the hidden surface problem by visiting the voxels in either back-to-front or front-to-back order. Splatting is a perfect example of an object-order algorithm in contrast to ray-casting, which is an image-order algorithm. For years, the splatting technique has been used to render volumes of various grid structures [Westover89, Mao96, Westover91]. Martyn introduced a novel approach to realistic real-time rendering scenes consisting of many affine IFS fractals [Martyn10]. The implementation based on splatting and hardware geometry instancing makes it possible to achieve efficient visualization with small memory requirements.

Another interesting work in the field of voxel visualization is the particle-based approach presented by Juckel and Beckhaus [Beckh07]. The authors developed visualization of 3D scalar field by using a particle system. They proposed a unique method for rendering complex shapes as fuzzy or diffuse objects inside virtual environments. The algorithm converts surface geometry into the voxel-like grid that specifies the appearance of the shape. Using GPU implementation, they achieved rendering of dynamic objects inside a voxelized surface geometry. Particle systems were designed to handle millions of simple objects perfectly characterize voxel structures.

Although all of the presented methods propose interesting ideas related to visualization of the voxels, only ray tracing approach is able to render realistic 3D objects. As we mentioned before, the SVO ray tracing approach is the current standard of voxel visualization. Advantages of this algorithm are scalability with the screen resolution and fixed rendering cost resulting from a constant ray count. We have developed a voxel rendering algorithm which offers similar advantages and uses the triangle rasterization pipeline.

Using a newly available 3D sparse texture and highly optimized ray marching approach, we perform filtering of visible voxels. Then we fill screen size voxel buffer with filtered voxels data. Finally, we render voxelized objects with triangle based pipeline. We have managed to achieve efficient rendering performance results with a fixed rendering cost.

3 VISUALIZATION ALGORITHM

In this section, we describe our approach to visualize voxelized objects with the use of geometry shaders and deferred rendering. The features of our algorithm are as follows:

- Efficient rendering of high-resolution voxelized objects.
- Support for both static and dynamic objects.
- Representation based on SVO.
- Minimization of memory consumption.
- GPU acceleration for geometry generation and rendering.

3.1 Voxel representation

Voxels are simply a 3D generalization of pixels. Each value on a regular grid stores information such as color, normal vector or density. One of the most significant disadvantages of voxels compared to polygons is the memory consumption for high-resolution grids. One of the possible solutions to deal with this issue is the use of SVO. It eliminates empty spaces. Moreover, it gives the hierarchical level of detail (LOD) information about the source object. Fig. 1 presents example of SVO structure visualization.

The simplest way to visualize a voxel is to render a cube in 3D space. In such a representation, one would need to store data for twelve indexed triangles with

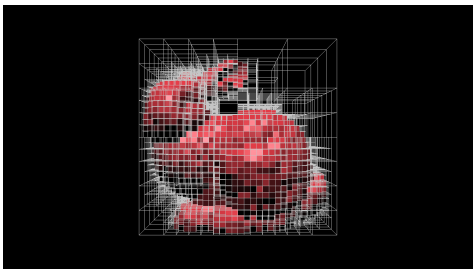

Figure 1: SVO structure created for sample object.

appropriate attributes. We could create proper data buffers on GPU and fill them with voxelized object data. Unfortunately, this is not an efficient solution for dynamic buffers. Also, we are still facing the problem with high memory consumption. Analyzing the rendering result of voxel representation with 3D cubes, we can realize that voxel looks same when viewed from different angles. Thus, we can visualize voxels as quads always faced to the viewer using billboarding approach [Behren05, Decaudin09].

Implementation based on quads instead of cubes significantly reduces memory requirements. However, it means that we need to use some kind of hardware geometry instancing or geometry generation method in order to render our object.

3.2 Geometry generation with shaders

Modern graphics APIs like OpenGL or DirectX offer a few ways to implement geometry generation with GPU programmable pipeline. The most straightforward way to implement our algorithm is to use geometry instancing functions. With this choice, we need to store our source quad in GPU memory and create data buffers for billboard attributes like positions and normal vectors. Geometry instancing is a very efficient method with data streaming functionality for handling dynamic objects. However, there is no way to control which quads will be generated and rendered on GPU. We can solve this by using streaming data buffers and performing calculations on CPU, but this is computationally expensive solution.

An alternative solution which we have implemented is based on the usage of programmable geometry shaders. In this method, we do not need quad representation data. We store our voxelized object as a collection of points and generate billboard quads on GPU using geometry shaders. Fig. 2 presents an example of rendering a 3D object based on a surface approximation with billboards.

As a result of using geometry shaders, we gain control on quads generation stage which, for example, gives us the possibility to execute the back-face culling algorithm or control billboards shape generation independently.

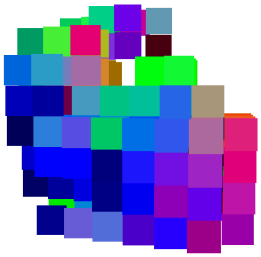

Figure 2: Visualization of low LOD object with collection of billboards.

3.3 Smooth shading realization

In the case of 3D objects based on polygonal representation, smooth shaded visualization can be achieved, for example, by using the Blinn-Phong normal interpolation model [Foley90]. Unfortunately, a voxelized surface has no information about adjacent voxels in the rasterization pass. This problem is typically solved by using the volume ray tracing algorithm which can be implemented on GPU with SVO structures. In order to achieve smooth shaded objects, we propose a new method based on the screen space approximation of voxel attributes data. Our smooth shading algorithm is based on multipass deferred rendering. In G-buffer generation pass we render voxel attributes data to floating point render targets.

The direct usage of rendered voxel attributes in the deferred composition pass with additional information about scene lighting produces blocky, flat shaded visualizations. In order to achieve smooth shaded visualization, we perform data interpolation by a filtering data texture with the 3x3 kernel Gaussian blur shader. Fig. 3 presents an example of voxel normal attributes interpolation in screen space.

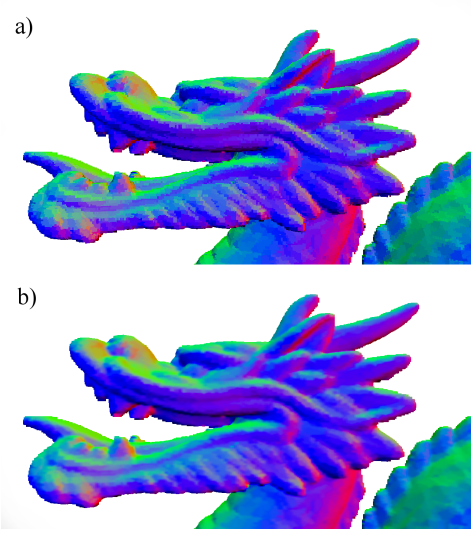

Figure 3: a) Object rendered without normal attribute interpolation b) Object rendered with normal attribute interpolation.

3.3.1 Smooth filtering control

In order to achieve a proper smoothing for all voxelized objects on the virtual scene, we must control the Gaussian blur intensity. The objects that are closer to the observer should be filtered stronger than the objects that are far away from the observer. Actually, we need to implement filtering control based on a similar manner as the LOD management algorithms [Lueb02].

In the proposed algorithm, we use the Gaussian blur with a 3x3 kernel as a voxel attribute filtering method. The most straightforward way to achieve stronger filtering is by changing the size of the blur kernel. However, it is a major waste of GPU computing resources, especially when we implement dynamic loops in our filtering shaders. We propose to use a filter shader sampler offset in order to achieve efficient and visually acceptable results.

The most commonly used LOD evaluation parameter is the distance [Lueb02]. We decided to use this parameter to control the smoothing intensity. Equation 1 presents a filter intensity calculation formula.

$$I = max\Big(0, min(\frac{MaxI}{(Dist - MinDist)^2}, MaxI)\Big) \qquad (1)$$

where:

I	= intensity of filter (sampler offset)
$MaxI$	= maximum intensity for the closest objects
$Dist$	= distance between object and observer
$MinDist$	= defined minimum distance

The result of the equation must be clamped to *<0, MaxI>*. Zero as a minimum value means that the object that is far away from the observer does not need to be filtered. The maximum intensity value must be defined by the user depending on virtual scene construction, as well as the minimum distance between an object and the observer, where voxel attributes are filtered with the maximum intensity.

3.3.2 Pixel depth based smoothing

Voxel attributes smoothing is done with screen space shaders. We cannot calculate the distance to each object on the virtual scene and pass it to the shader as a uniform value. Also, we cannot perform multiple filtering passes in order to fit in the time requirements of real-time graphic engines. In order to perform a distance-based data filtering, we use depth a buffer from G-buffer pass.

Depth buffer stores the depth of a generated pixels. In order to perform a smoothing operation with equation 1, we need to calculate a pixel position in world space. Using the inverse of the view projection matrix, we can reconstruct the pixel position in world space [Wright10]. By using the obtained value with a camera position transferred to shader code as a uniform, we are able to use the proposed equation and achieve a proper, distance based voxel attribute smoothing.

4 SCREEN SPACE BILLBOARD VOXEL BUFFER

In this section, we describe our approach to select voxel render candidates from an object voxel point cloud data in order to render the high-resolution object with a fixed size data buffer.

In order to limit the number of voxels required for rendering, we need to filter the voxel point cloud that would fill the screen space data buffer. Filtering operation can be done in a wide selection of methods. Voxel point cloud can be projected on the screen and by using depth test, render candidates can be selected. Another possible solution is to perform hierarchical occlusion queries in order to find visible SVO nodes. However, our main goal was to develop an algorithm with screen space computation complexity which will offer comparable performance results as the ray tracing. In order to select render candidates we developed ray marching algorithm with 3D lookup texture.

4.1 Filtering algorithm components

The smallest part of the screen is one pixel. It means that the image covers the maximum information when all pixels are filled by exactly one independent voxel. However, a 3D object can be represented by much more voxels than is needed to fill a render target. It is the biggest disadvantage of the graphics representation with polygons. Vertex shaders can be invoked for the data that would not fill any result image pixels or pixel overdrawn can cause an enormous performance hit. This problem does not occur in the context of rendering with ray tracing approach. For this reason, we decided to develop an efficient way to select object voxels that would fill the resulting image.

Our algorithm uses the structure which we called Screen Space Billboard Voxel Buffer. The algorithm is based on the three components:

- **Screen Space Billboard Voxel Buffer** — a fixed size data buffer which contains voxel data used in the rendering process. The size of the buffer corresponds to the render target resolution. Voxel filtering algorithm selects render candidates and fills voxel buffer data.
- **Voxel point cloud** — voxels data from the object's selected LOD. Each point stores information such as position, normal vector, texture coordinates and optional object id.

- **Sparse Lookup Texture** — 3D texture for voxel point cloud lookup table. With ray marching approach, we get an id of render candidate voxel. Using that id, voxel data is copied from voxel point cloud buffer to screen space voxel buffer. Thanks to the modern graphics APIs, we can create the texture that is much bigger than available memory and fill only selected *pages* of the texture [Wright10]. Object LOD control is managed by using sparse texture mipmaps to store 3D lookup table for different object LODs.

4.2 Voxel Filtering algorithm

In this section, we describe base steps of voxel filtering algorithm. The developed algorithm is based on the 3D sparse lookup texture ray marching.

Using ray intersection test with 3D texture, we are able to efficiently filter voxel cloud. However, this approach creates two potential problems. Firstly ray marching requires a lot of texture sampling operations. In order to achieve efficient performance results, it is required to implement a few optimization techniques like Object Order Empty Space Skipping [RezkSal09, Vidal08]. Secondly, standard 3D texture will require a lot of the VRAM memory. We solve this problem with the newly developed sparse texture from modern graphics APIs [Wright10]. From OpenGL 4.4 specification, AMD sparse texture extensions developed by Graham Sellers is available by GL_ARB_sparse texture. Additionally, OpenGL 4.5 specification added a new version of this extension with full shader side control.

4.2.1 Algorithm preparation steps

Voxel filtering algorithm can be divided into the data preparation and execution steps. The preparation steps are as follows:

1. Extract voxel points clouds from the required LODs. This step can be done in precomputation pass on virtual scene initialization.
2. Create 3D lookup table for voxel point cloud and store it in sparse texture. If we need to use LOD management of virtual scene, we store additional lookup tables in sparse texture mipmaps.
3. Create simplified 3D object triangle mesh based. This mesh will be used to optimize ray marching operation. It is important to create a polygonal mesh that vertices positions are in range of <-1.0,1.0>.

4.2.2 Algorithm execution steps

In the application rendering loop we perform algorithm execution steps as follows:

1. Perform visibility test of the 3D object with frustum culling and optionally occlusion culling tests. If our object is not visible or it is occluded by another object, the algorithm ends here for the selected object.
2. Render all visible objects simplified mesh offscreen. For all objects, we save normalized object space position in first render target and the object world space position in the second render target. The first texture will be used to perform Object Order Empty Space Skipping. The second will be used to handle scenes with numerous objects. Additional object id or LOD information will be saved in the same textures if needed.
3. Perform ray marching for all pixels using obtained textures as input. Using 3D sparse lookup texture, find first intersection and store voxel data in Screen Space Billboard Voxel Buffer. It is important to clear the current pixel screen space buffer data from the last frame in order to optimize rendering pass.
4. Render Screen Space Billboard Voxel Buffer using the algorithm described in section 3.

4.2.3 Algorithm conclusion and limitation

Using the fixed size voxel buffer we limited vertex shader invocations to the fixed number. Moreover, the voxel filtering algorithm based on 3D sparse texture ray marching can be implemented in a very efficient way. Thanks to that, we can render a virtual scene with numerous high-resolution 3D objects with triangle rasterization pipeline in real-time. It is impossible without filtering step with today's hardware.

The fixed size voxel buffer is a particularly efficient method in the case of high-resolution 3D objects. If an object is represented by more voxels that can be stored in Screen Space Billboard Voxel Buffer, the draw call operation can be significantly optimized by limiting vertex shader invocations. For example, Stanford Bunny object on 9 level of SVO is represented by about 2.5 million voxels. After the filtering pass, we need only about 140 thousand voxels to render the object. It means that even with the additional filtering pass and fixed cost rendering operation, performance results is noticeably better than rendering object without filtering pass.

5 IMPLEMENTATION DETAILS

In this section, we describe important implementation details of our algorithm. We have implemented our method using OpenGL 4.5 API with C++14 but there are no limitations to using any other graphics interface or programming language. All included shader source code listings are prepared in GLSL language. Due to the simplicity of the billboard based voxel representation, the presented algorithm can be easily implemented and integrated into all popular game engines. The only requirement is the support for programmable geometry and compute shaders.

5.1 Screen Space Billboard Voxel Buffer preparation implemenation

In this section, we describe *Screen Space Billboard Voxel Buffer* implementation. We will focus mainly on ray marching extensions and algorithm optimization steps.

5.1.1 Screen size static vertex buffer

The base component of the developed Screen Space Billboard Voxel Buffer algorithm is a static, fixed size data buffer for voxels data. Due to that we can create a static Vertex Buffer Object and fill it using compute shaders. Using Shader Storage Buffer Objects introduced in OpenGL 4.X API, we can bind the Vertex Buffer Object and access it from the compute shader. Therfor, both voxel point cloud input data and output Screen Space Billboard Voxel Buffer can be accessed on GPU. The layout of vertices data is the same as vertex layout that we use in the voxel visualization algorithm. The only potential difference is additional object id information if our virtual scene contains many independent 3D objects. In that case, an additional material data array is necessary to handle shading in rendering pass.

5.1.2 3D sparse texture

In our implementation, we used GL_ARB_sparse_texture and GL_EXT_sparse_texture2 extension according to test hardware specification. For a lookup table, *R32UI* internal format texture was used to store voxels id in the red channel. According to the texture internal format, the sparse texture is dived to the specified number of pages. We used 16x16x16 size pages for lookup table texture. If some page is empty, GPU will not allocate video memory for that page. On the shader side, we can check if sampled data is committed or not. If we cannot use the latest version of sparse texture extensions, according to the driver specification, sampling operation should return zeros.

5.1.3 Voxel filtering implementation

Voxel filtering algorithm was implemented with compute shader. Using ray marching approach, we seek for render candidates and then copy voxel data from point cloud to screen space voxel buffer. In order to optimize ray marching step, Object order empty space skipping pass was implemented. Using a prepared simplified triangle mesh, we render world and object space positions to off-screen render targets. Using saved pixel object space position, we can optimize ray marching by starting marching very close to the object surface. Additionally, we optimize ray marching for big, empty spaces. Figure 4 presents render results of example scene.

Created 3D sparse texture and pre-pass rendering results are used in final voxel filtering step. Listing 1 presents main parts of voxel filtering compute shader code. Ray marching and data preparation code have been omitted. Additionally, in order to simplify listing, some code was reduced to pseudocode or comments.

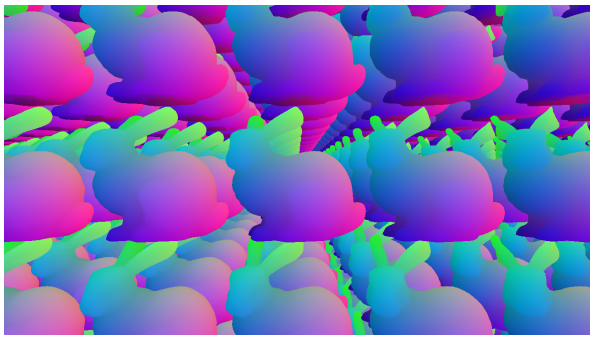

Figure 4: Ray marching start position from object order empty space skipping optimization pass.

```
layout (rgba16f) uniform image2D worldSpaceTex;
layout (rgba16f) uniform image2D objectSpaceTex;
uniform usampler3D lookupTex;

struct VertexLayout {
  vec3 position;
  vec3 normal;
  vec2 uv;
};

layout(std140, binding = 0) buffer VBO_Input {
  VertexLayout vbo_in[];
};

layout(std140, binding = 1) buffer VBO_Output {
  VertexLayout vbo_out[];
};

void main(void) {
  int id = pix.x*size.y + pix.y;
  vbo_out[id].uv.x = -1.0; // clear SSBVB

  vec4 posStart = imageLoad(objectSpaceTex, pix);
  if(IsEmpty(depth.a)) return;

  // perform ray marching fo defined sample count
  uint sample = texture(lookupTex,pos).r;
  if(IsResident(sample)) {
    int voxel = int(sample);
    vec4 off = imageLoad(worldSpaceTex, pix);
    // init SSBVB
    vbo_out[id].position = vbo_in[voxel].position;
    vbo_out[id].position += off.xyz;
    vbo_out[id].normal = vbo_in[voxel].normal;
    vbo_out[id].uv = vbo_in[voxel].uv;
  }
```

Listing 1: Voxel filtering compute shader.

5.2 Voxel rendering implementation

The major disadvantage of voxel representations is memory consumption. Using 3D textures to store object data is a major waste of the VRAM. For example, efficient usage of allocated memory in the case of Stanford Bunny surface voxelized 256^3 resolution is about 1.5%. The usage of SVO solves this problem. The SVO is a great method of voxel data compression based on optimizing empty and constant spaces. Additionally, we automatically gain an object LODs collection.

Ray tracing is the popular method of SVO rendering. It is efficient and produce great rendering results. However, in that method, full SVO data must be stored in the VRAM or data must be streamed from RAM to GPU memory. In order to easily handle the virtual scenes with many different objects based on the SVO, we used a different approach. Listings 2 - 4 presents a voxel rendering pipeline with geometry shaders.

```
layout (location = 0) in vec3 pos;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

uniform mat4 modelView;
uniform mat3 invTModelView;

out VertexData {
  vec3 normal;
  vec3 position;
  vec2 texCoord;
} VertexOut;

void main() {
  gl_Position = vec4(pos,1.0);
  vec4 viewPos = modelView*vec4(pos,1.0);
  VertexOut.normal = invTModelView*normal;
  VertexOut.position = viewPos.xyz;
  VertexOut.texCoord = texCoords.xy;
}
```

Listing 2: Voxel rendering vertex shader

```
layout (points) in;
layout (triangle_strip, max_vertices = 4) out;

uniform mat4 projection;
uniform mat4 modelView;
uniform vec3 size;
uniform vec3 cameraPosition;

in VertexData {
  vec3 normal;
  vec3 position;
  vec2 texCoord;
} VertexIn[1];

out VertexData {
  vec3 normal;
  vec3 position;
  vec2 texCoord;
} VertexOut;

void main() {
  if(Backface() || EmptyScreenBuffer())
    return;

  vec4 center = modelView * gl_in[0].gl_Position;

  gl_Position = projection * (center + size);
  VertexOut.position = VertexIn[0].position;
  VertexOut.normal = VertexIn[0].normal;
  VertexOut.texCoord = VertexIn[0].texCoord;
  EmitVertex();
  // ... same operation for the rest
  EndPrimitive();
}
```

Listing 3: Voxel rendering geometry shader

```
uniform sampler2D tex;

in VertexData {
  vec3 normal;
  vec3 position;
  vec2 texCoord;
} VertexIn;

out vec4 albedoOutput;
out vec4 posOutput;
out vec4 normOutput;

void main() {
  vec3 norm = normalize(VertexIn.normal.xyz);
  albedoOutput = texture(tex, VertexIn.texCoord);
  posOutput =
       vec4(VertexIn.position,LinearizeDepth());
  normOutput = vec4(norm, 1.0);
}
```

Listing 4: Voxel rendering pixel shader

5.3 Screen space attributes smoothing integration

With developed algorithm, we are able to render a voxelized object alongside triangle objects and use any triangle rasterization pipeline algorithm. Unfortunately, a challenge appears when we need to implement the screen space data smoothing for selected voxel attributes. Using the deferred rendering pipeline we store all frame normal data in the G-buffer. Smoothing cannot affect triangle based objects because the information will be lost.

We developed an integration method based on stencil buffer. Using the stencil test we can exclude triangle-based objects from voxel objects. The algorithm based on stencil buffer is as follows:

1. Create and attach stencil buffer to the G-buffer.
2. Setup stencil to write defined value to stencil when rendering triangle objects and a different value for the voxel objects.
3. Attach stencil buffer to filtering pass frame buffer.
4. Setup stencil test to pass only fragments related to the triangle objects.
5. Setup stencil test to pass only voxel objects and apply filtering shader to passed fragments.
6. Use smoothed attributes in deferred composition pass.

6 RENDERING AND PERFORMANCE TEST RESULTS

All depicted timings were obtained on Intel Core i5 2500K CPU with NVidia GeForce GTX 660 GPU. All algorithms were implemented using OpenGL 4.5 API with C++14 for Windows 10 64-bit. We used Stanford Repository models as a test object [Stanford11]. Table 1 presents performance, memory requirements and test results for static and dynamic streamed voxel objects. Figures 5 - 7 presents rendering results of proposed voxel rendering algorithm.

Without a doubt, today's standard and most studied method for visualizing SVO is based on using the ray tracing algorithm on GPU. For that reason, we performed performance tests and compared them with our algorithm. As a reference, we used the „Efficient

sparse voxel octrees" implementation that is available online [Laine10]. Performance test results are presented in Table 2. The results show that our approach offers comparable or even better performance than ray tracing SVO visualization.

It order to test how our algorithm performs on modern hardware designed for computer games, we performed additional tests on PC with Intel Core i7 4790K CPU with NVidia GeForce GTX 980 GPU. We prepared objects voxelized in 2048^3 resolution. Figure 8 presents rendering results of high-resolution objects. Figure 9 presents performance test of the scene rendered with Screen Scene Billboard Voxel Buffer.

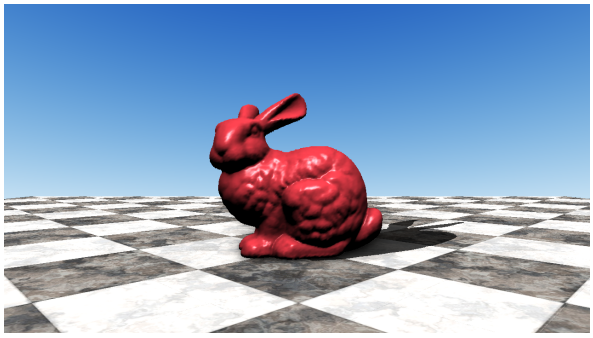

Figure 5: Stanford Bunny, 9 octree level.

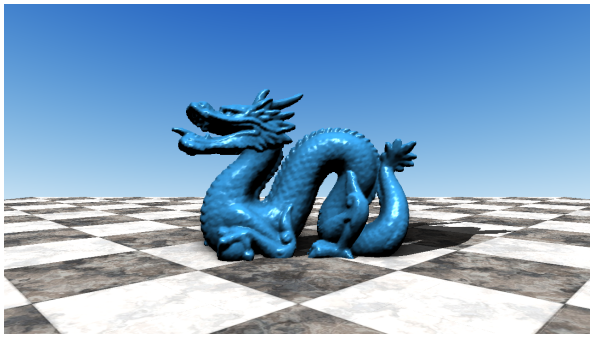

Figure 6: Stanford Dragon, 9 octree level.

Figure 7: Stanford Lucy, 10 octree level.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach to efficient real-time rendering of numerous high-resolution

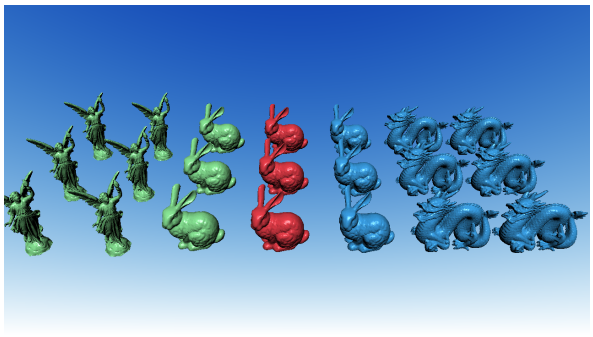

Figure 8: 21 test objects, 402 million voxels, 22 FPS achieved on 720p render target with GeForce GTX 980.

voxelized objects with the fixed size *Screen Space Billboard Voxel Buffer*. The developed method can be used to render 3D objects represented by SVO with a standard triangle-based pipeline graphics engine. Thanks to the limitation of vertex shader invocations and the geometry shaders usage it is possible to achieve real-time rendering of billions of voxels. We achieved comparable or even better rendering performance results in comparison with the SVO ray tracing approach Moreover, our method is applicable to render both static and dynamic objects in real-time with the full support of modern hardware graphics APIs and rendering algorithms.

Used *Object Order Empty Space Skipping* efficiently optimized 3D sparse texture sampling. However, the current implementation is highly optimized for rendering non-occluding objects. If we render the scene with many occluding objects, simplified triangle meshes causes small artifacts on the object's edges. We need to extend our ray tracing implementation with additional ray traversal for occluded pixels.

An obvious step forward would be an implementation of SVO traversal as a substitute for ray marching filtering approach. We can actually use sparse textures to store SVO. In that case, we will need to store full SVO in GPU memory. It will increase memory requirements and slightly impends dynamic object handling. However, voxel filtering could be faster and more precise in comparison with ray marching approach.

8 REFERENCES

[Bau11] Bautembach D., Animated sparse voxel octrees, Bachelor Thesis, University of Hamburg, 2011.

[Beckh07] Beckhaus S., and Juckel, T., Rendering diffuse objects using particle systems inside voxelized surface geometry, The 15-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision2007 (WSCG 2007).

[Behren05] Behrendt, S., Colditz, C., Franzke, O., Kopf, J., and Deussen, O., Realistic real-time

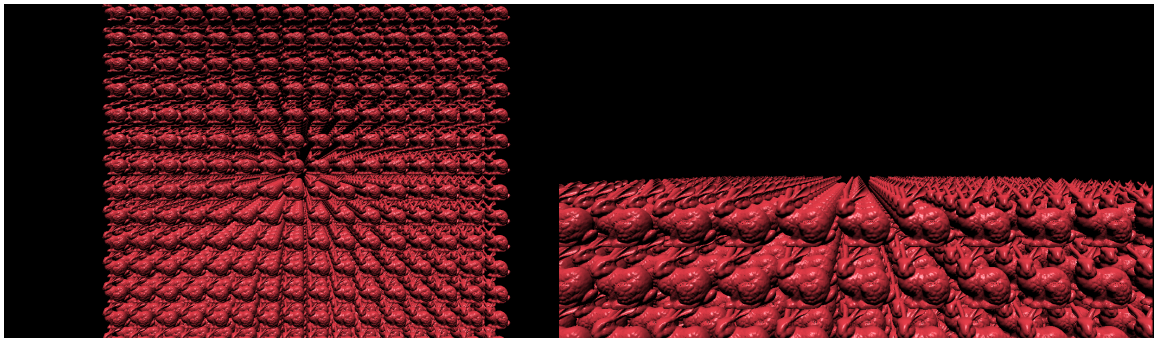

Figure 9: Rendering results using Screen Space Billboard Voxel Buffer in 720p with Nvidia Geforce GTX 980. Test scene contains 4096 Stanford Bunny which are represented by about 10 billion voxels (20 billion triangle in base algorithm). We achieved about 70 FPS for the left image and about 160 FPS for the right image.

Object	Octree level	Static object rendering time	Streamed object rendering time	Voxel grid file	Octree file	VRAM
Bunny	7	0.07 ms	0.11 ms	2.06 MiB	15.0 MiB	0.67 MiB
	8	0.26 ms	0.61 ms	8.30 MiB	60.9 MiB	0.67 MiB
	9	1.24 ms	3.75 ms	33.20 MiB	255 MiB	3.04 MiB
Dragon	7	0.10 ms	0.12 ms	2.81 MiB	22.2 MiB	0.92 MiB
	8	0.37 ms	0.95 ms	11.3 MiB	88.8 MiB	0.92 MiB
	9	1.81 ms	5.15 ms	45.3 MiB	363 MiB	4.26 MiB
Lucy	7	0.05 ms	0.11 ms	1.69 MiB	11.0 MiB	0.52 MiB
	8	0.20 ms	0.40 ms	6.86 MiB	45.3 MiB	0.52 MiB
	9	0.95 ms	2.73 ms	27.5 MiB	183 MiB	2.32 MiB

Table 1: Performance, memory requirements and test results for static and dynamic streamed voxel objects without using developed *Screen Space Billboard Buffer* algorithm. Render target resolution was 720p.

Object	Octree level	Resolution	Ray tracing render time	Ray tracing manage time	Ray tracing FPS	SSBVB render time	SSBVB manage time	SSBVB FPS
Bunny	7	720p	8.43 ms	10.20 ms	78	2.61 ms	1.09 ms	253
		1080p	15.16 ms	17.90 ms	48	6.80 ms	2.48 ms	102
	8	720p	8.80 ms	10.60 ms	76	2.56 ms	1.46 ms	235
		1080p	15.25ms	18.30 ms	47	5.92 ms	3.05 ms	106
	9	720p	9.32 ms	11.25 ms	72	2.59 ms	2.34 ms	192
		1080p	16.01 ms	19.21 ms	44	5.92 ms	4.63 ms	90
Dragon	7	720p	6.44 ms	8.20 ms	93	2.88 ms	1.96 ms	198
		1080p	11.10 ms	13.80 ms	60	7.64 ms	3.89 ms	83
	8	720p	6.89 ms	8.60 ms	89	2.82 ms	3.38 ms	153
		1080p	11.68 ms	14.20 ms	68	6.52 ms	6.74 ms	73
	9	720p	7.47 ms	9.22 ms	84	2.85 ms	5.78 ms	112
		1080p	11.92 ms	15.30 ms	54	6.50 ms	11.91 ms	54
Lucy	7	720p	4.51 ms	6.20 ms	115	2.59 ms	1.17 ms	252
		1080p	8.01 ms	10.50 ms	72	6.65 ms	2.38 ms	107
	8	720p	4.75 ms	6.45 ms	112	2.51 ms	1.86 ms	219
		1080p	8.30 ms	10.98 ms	69	5.67 ms	3.70 ms	101
	9	720p	5.04 ms	6.71 ms	109	2.49 ms	3.11 ms	170
		1080p	8.70 ms	11.30 ms	69	5.63 ms	6.13 ms	81

Table 2: Performance test comparison between developed *Screen Space Billboard Buffer* algorithm and ray tracing implementation [Laine10]. SVO ray tracing times are obtained from the performance tools included in the implementation.

rendering of landscapes using billboard clouds, Computer Graphics Forum, 2005.

[Crassin11] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E., Interactive indirect illumination using voxel cone tracing, Computer Graphics Forum (Proceedings of Pacific Graphics 2011), vol. 30, no. 7, sep 2011.

[Decaudin09] Decaudin, P., Neyret, F., Volumetric Billboards, Computer Graphics Forum, Volume 28 (8), pp 2079-2089, 2009.

[Foley90] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F., Computer Graphics: Principles and Practice (2Nd Ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[Ju02] Losasso, T. Ju, F., Schaefer, S., and Warren, J., Dual contouring of hermite data, ACM Trans. Graph., vol. 21, no. 3, pp. 339-346, Jul. 2002.

[Laine10] Laine, S., and Karras, T., Efficient sparse voxel octrees, in Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ser. I3D 2010. New York, NY, USA: ACM, 2010, pp. 55-63.

[Lueb02] Luebke D., Watson B., Cohen, J., D., Reddy, M., and Varshney, A., Level of Detail for 3D Graphics. New York, NY, USA: Elsevier Science Inc., 2002.

[Lorensen87] Lorensen, W. E., and Cline, H. E., Marching cubes: A high resolution 3d surface construction algorithm, SIGGRAPH Comput. Graph., vol. 21, no. 4, pp. 163-169, 1987.

[Mao96] Mao, X., Splatting of Non Rectilinear Volumes Through Stochastic Resampling, IEEE Transactions on Visualization and Computer Graphics, 2(2), 1996, pp. 156-170.

[Martyn10] Martyn T., Chaos and graphics: Realistic rendering 3d ifs fractals in real-time with graphics accelerators, Comput. Graph., vol. 34, no. 2, pp. 167-175, Apr. 2010.

[Potts04] Potts S., and Möller T., Transfer functions on a logarithmic scale for volume rendering, in Graphics Interface 2004, ser. GI 2004, 2004, pp. 57-63.

[Reinders07] Reiders, J., Intel Threading Building Blocks, O'Reilly Associates, Inc., 2007.

[RezkSal09] Rezk Salama, C., Hadwiger, M., Ropinski, T., Ljung, P., Advanced Illumination Techniques for GPU Volume Raycasting, ACM SIGGRAPH Courses Program, 2009.

[Stanford11] The Stanford 3D Scanning Repository, Stanford University, 22 Dec 2010, Retrived 17 July 2011.

[Vidal08] Vidal, V., Mei, X. Decaudin, P., Simple Empty-Space Removal For Interactive Volume Rendering, J. Graphics Tools, vol. 13, no. 2, pp. 21-36, 2008.

[Westover89] Westover, L. Interactive volume rendering. In: VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization, New York, NY, USA, ACM (1989) 916

[Westover91] Westover, L.A., SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm, Ph.D. Dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill, 1991.

[Wil13] Willcocks, C. G., Sparse volumetric deformation, Ph.D. disseration, Durham University, 2013.

[Wright10] Wright, R., S., Haemel, N., Sellers, G., Lipchak, B., OpenGL SuperBible: Comprehensive Tutorial and Reference, Addison-Wesley Professional, 2010.

Last page should be fully used by text, figures etc. Do not leave empty space, please.

Do not lock the PDF – additional text and info will be inserted, i.e. ISSN/ISBN etc.